Elección de Caminos de Acceso en Optimizadores de Bases de Datos - Un Lenguaje de Patrones

HORACIO PEÑAFIEL – Universidad Nacional de La Plata

Abstract

El subsistema de optimización de consultas SQL es parte hoy en día de todo sistema de administración de bases de datos relacional (RDBMS). Dada una petición al servidor, resolver el problema de la elección de un "*Camino de Acceso"* ("*Access Path*" en inglès) es de gran importancia para dicho componente, ya que se trata de seleccionar una serie finita de pasos para retornar un conjunto de tuplas (filas) de una o más relaciones (tablas o vistas). Con este objetivo, el optimizador cuenta con algoritmos, heurísticas, índices, y otras técnicas más puntuales.

En este trabajo, enunciaremos un Lenguaje de Patrones para resolver este problema. Para ello, nos basaremos en la teoría específica acerca del tema, así como en las diversas implementaciones que los distintos RDBMSs existentes en el mercado hicieron al respecto. Tal como lo hace el mismo optimizador, no pretenderemos obtener una única respuesta óptima, sino que buscaremos guías que nos ayuden a comprender la arquitectura y el diseño de esta parte tan importante de los optimizadores de SQL.

Categories and Subject Descriptors: D.2.11 **[Software Engineering]** Software Architectures–Patterns → **[Databases]** Performance and Optimization
General Terms: Databases; Access Paths; Patterns
Additional Key Words and Phrases: Optimizer, Performance, Architectural Patterns

Patrón: Elector de Caminos de Acceso

Problema:

Cuando un Servidor de Bases de Datos Relacional recibe una consulta SQL, se desea saber qué pasos realiza para devolver el resultado correcto, con el menor costo, en un tiempo razonable, y haciendo el uso más eficiente de los recursos que cuenta.

Fuerzas:

Se trata de un problema de optimización de recursos, los cuales en el caso de un motor de bases de datos son: **CPU, Memoria y Disco (I/O)**. Nos encontramos que, en la práctica, al intentar optimizar el uso de uno de estos elementos, incurrimos en un mayor costo de uso de alguno de los restantes.

De los tres mencionados arriba, el acceso y transferencia de datos de y hacia disco es el que sea más notorio como posible cuello de botella. A nivel físico, cuestiones como la latencia (movimiento del cabezal para una búsqueda de un dato en particular), y la velocidad de los canales de acceso

(transferencia de datos), pueden afectar en forma directa a la performance de un servidor de bases de datos. Sin embargo, este almacenamiento secundario nos brinda la ventaja de que el espacio disponible es ampliamente mayor que el que disponemos en Memoria o en los registros internos de la CPU. Para ciertas operaciones, será necesario recurrir al disco, no sólo como almacenamiento definitivo, sino también como un espacio temporal donde se pueda leer y escribir información de transacciones que están siendo procesadas (buffers). Del recurso "Disco" nos interesará observar el **Tiempo de Acceso (Latencia).**

En cuanto a la Memoria (almacenamiento primario), nos encontramos con que la velocidad de acceso es muy superior a la del disco (aunque no tanto como los registros y/o caché del procesador). Debido a que es un recurso crítico y escaso en el servidor, sólo se almacena permanentemente un conjunto limitado de páginas de datos e índices en los buffers. El riesgo es saturar al equipo provocando que dichas páginas se deban bajar a disco temporalmente para liberar espacio, y luego volver a subirlas a memoria cuando sea necesario. Este fenómeno, amplificado en forma inversamente proporcional a la cantidad de memoria libre del sistema, se denomina **"trashing"**, y se da no sólo en los sistemas de bases de datos sino que también aparece a nivel de los sistemas operativos con memoria virtual – el resultado observable es que la mayor parte del tiempo de procesamiento se pierde subiendo y bajando páginas de memoria a disco y viceversa. Asimismo, no debemos olvidar el importante hecho de que el mismo DBMS es una aplicación en sí misma, y que los módulos ejecutables requieren espacio adicional donde residir. Por lo tanto, será de nuestro interés optimizar el **Espacio Utilizado en Memoria** (como puntualizamos anteriormente, menos memoria libre aumenta el riesgo de "trashing"), así como reducir los **Tiempos de Alocación** (el tiempo que se requiere para obtener un bloque de memoria).

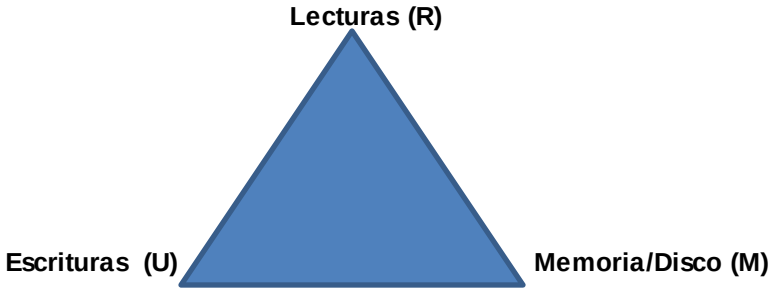
Finalmente, debemos decir que el primer recurso, la CPU, puede ser un cuello de botella debido a varias causas. Los sistemas de bases de datos aplican en forma recurrente un amplio conjunto de algoritmos que son procesador-intensivos. Si bien los procesadores cuentan con almacenamiento primario adicional (en forma de caches de primer nivel), nos interesa tomar como medida de performance de la CPU el **Tiempo de Procesamiento**.

La conjetura RUM

En su trabajo **"The RUM Conjecture (La conjetura RUM)",** [Athanass01] proporciona un enfoque alternativo al que detallamos arriba, basándose en esta hipótesis:

"El desafío que enfrenta el Optimizador … se basa en minimizar: 1) los tiempos de lectura (R); 2) los costos de actualizar los datos (U); y 3) el espacio en memoria requerido (M). Se plantea que al optimizar dos de éstos factores, se obtiene un impacto negativo sobre la performance del tercero."

Basándonos en este trabajo, podemos graficar la interacción de las Lecturas, Escrituras y Memoria (RUM) como dimensiones que deben estar balanceadas:



Algunos ejemplos.

- Si se trata de optimizar la velocidad de las lecturas (R), podemos agregar un Indice B+-Tree. Este cambio requiere sin embargo un costo adicional al actualizar los datos (U), ya que se deberá escribir tanto en las páginas de datos como en las del nuevo índice. Asimismo, dicha estructura de árbol tiene que mantenerse en disco y/o en memoria para ser utilizado (M).
- Si queremos optimizar para el uso de poca memoria, podemos calcular información en lugar de almacenarla (M). Esto ocasiona una carga mayor a la CPU (R), y posiblemente al disco (U).
- Otra forma de optimizar las lecturas (R) es replicando o particionando la información. En el primer caso, tenemos accesos más rápidos a un costo de datos redundantes (U/M), mientras que al particionar nos encontramos con una carga a la CPU (U) a la hora de administrar dicho sistemas distribuídos.

En resumen, podemos decir que no hay ***"balas de plata"***. En lo que sigue de este trabajo, trataremos de enfocar estos problemas mediante una arquitectura flexible del Optimizador de SQL.

Solución:

Se implementa un componente en el Optimizador del RDBMS denominado **"Optimizador de Caminos de Acceso"**. Llamamos **"Camino de Acceso"** a una serie de pasos concretos que el Optimizador ejecuta sobre el Motor de Almacenamiento, con el fin de obtener un conjunto de tuplas (filas) de una o más relaciones (tablas o vistas), con el menor costo posible, y haciendo uso eficiente de los recursos disponibles. Para realizar esta tarea, cuenta con información acerca de la estructura de la base de datos, así como estadísticas de uso de cada componente de la misma. También recibe de otro componente, llamado **"Algebrizer"**, un árbol de sintaxis que representa la consulta a ejecutar, obtenido a partir del SQL original.

El resultado de la ejecución del Optimizador será un **"Plan de Ejecución Fìsico" o "Camino de Acceso"**, que le dicta al Motor de Almacenamiento la serie de pasos y estrategias a utilizar para llevar a cabo la ejecución física de la petición. Es importante aclarar en este punto, que dicho Plan de Ejecución en sí también consume recursos, por lo que la tarea de Elección del Camino de

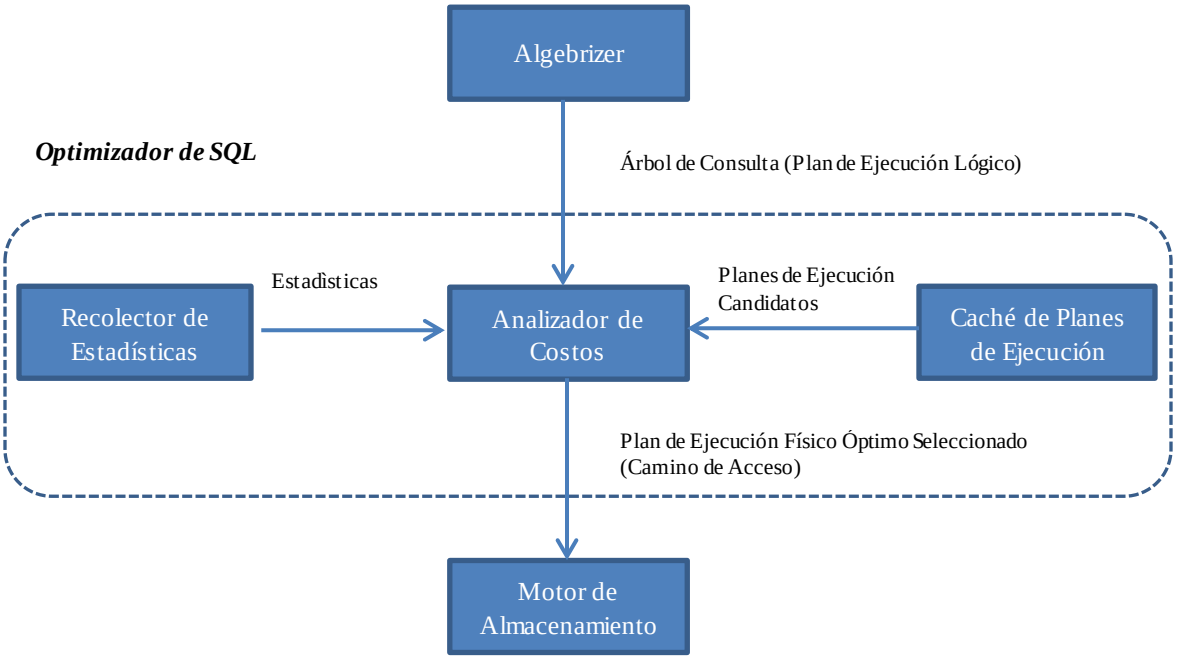
Acceso, se debe hacer en forma automática, en un tiempo razonablemente rápido, y con un costo mínimo agregado.

Finalmente, debemos considerar que si se requiera que el DBA o la misma aplicación cliente balanceen manualmente el uso de recursos requeridos, podemos encontrarnos con que esta tarea es bastante tediosa y no siempre conducente a los resultados deseados, ya que la elección del Camino de Acceso óptimo (de menor costo), depende en gran medida también de la consulta a ejecutar. De aquí se deriva la necesidad de que dicho procedimiento de optimización quede bajo la responsabilidad del Optimizador de SQL.

Arquitectura:

El problema de **"Optimización de Caminos de Acceso"** se planteó ya en los primeros sistemas de bases de datos experimentales. En particular, el DBMS denominado **"SYSTEM-R"**, creado por IBM en la década de 1970, es considerado tanto como el primer sistema relacional de Bases de Datos, como el precursor de la arquitectura que estamos discutiendo. [Selinger01]

Podemos bosquejar la arquitectura de un Optimizador según el siguiente modelo (delimitado por líneas de puntos):



El Optimizador recibe del componente **"Algebrizer"** una representación binaria del árbol de sintaxis, producto de haberse parseado la sentencia SQL, denominado **"Arbol de Consulta"**. Una vez obtenido dicho Árbol de Consulta, el Optimizador procede a consultar al **"Recolector de Estadìsticas"**. Este componente, periódica y automáticamente, almacena en su propia base de datos mediciones tales como la cantidad de tuplas procesadas, índices utilizados, parámetros, etc.

Esta información es vital para el Optimizador, ya que muchas de sus decisiones dependerán del grado de conocimiento que posea de los datos a procesar.

Una vez obtenidas las Estadísticas relevantes, el Optimizador construye un conjunto de **"Planes de Ejecución Candidatos"**. Estos Planes de Ejecución (lógicos) comparten entre sí el objetivo de dar resultado a la petición SQL original. Sin embargo, el **"Camino de Acceso"** (físico) propuesto por cada uno de ellos puede ser significativamente distinto.

La elección de un Plan de Ejecución (y su Camino de Acceso asociado) se hace mediante un **"Análisis de Costos"**. Básicamente, se comparan los costos de una posible ejecución de cada Plan Alternativo, buscando el de menor impacto.

Finalmente, una vez elegido el **"Camino de Acceso Óptimo"**, el Optimizador pasa dicho Plan de Ejecución al **"Motor de Almacenamiento"**, que es el encargado de ejecutar los operadores seleccionados en la secuencia correcta. Asimismo, dicho Plan de Ejecución se almacena en el **"Caché de Planes de Ejecución"** para una posible consulta del mismo en el futuro.

Patrones Relacionados:

Un **"Elector de Caminos de Acceso"** cumple las funciones de **"Intérprete"** [Gof01], siendo su entrada el Árbol de Consulta, provisto por el componente **"Algebrizer"**, y su salida un **Plan de Ejecución** que implementa uno o más **Caminos de Acceso**.

Asimismo, la representación de dichos **"Caminos de Acceso"** se asemeja al Patron **"Composite"**, donde el objeto compuesto estaría dado por el **"Plan de Ejecución"**, y los componentes serían los **"Operadores del Plan de Ejecución"**.

***

Patrón: Recolector de Estadísticas

Problema:

La elección de un Camino de Acceso depende en gran medida del conocimiento que tenga el **Optimizador** de los datos que se van a procesar, ya que esta información impacta de forma directa en la eficiencia de estimar los costos asociados a una consulta. Se requiere información adicional acerca de los datos en cada conjunto de datos, para poder computar eficientemente indicadores como la cardinalidad y la selectividad de los mismos al responder a un conjunto de consultas SQL.

Fuerzas:

Las **Estadísticas** son información adicional a los datos en una base de datos. Si prescindimos de dicha información, el Optimizador puede degradar la performance en conjunto al intentar calcular en tiempo real toda la información que necesita para generar sus Planes de Ejecución, o directamente no poder elegir un **Camino de Acceso**. Por otra parte, un trabajo en exceso del Recolector de Estadísticas, implicará también un mal uso de los recursos disponibles (sobre todo CPU y memoria). Ambos extremos serían ciertamente la causa de una degradación considerable en la performance del servidor SQL.

Solución:

Se implementa un componente, denominado **"Recolector de Estadísticas"**, que almacenará información en paralelo a las transacciones ordinarias de una base de datos. Dichas **Estadísticas** se basarán en el comportamiento de **Poblaciones (Relaciones o Tablas)**, de las cuales se derivan **Muestras (Tupas o Filas)**. De cada **Muestra** tendremos **Variables (Atributos o Columnas)**.

El **Recolector de Estadísticas** es un proceso que se ejecuta en segundo plano, y que construye una estructura de datos denominada **"Histograma"**. Un **Histograma** es una tabla que resulta de dividir el rango de valores de una variable en intervalos o **"buckets"**, y calcular la frecuencia en que los valores de dicha variable caen en cada "bucket". Esto se hace mediante un muestreo periódico de las transacciones que se van realizando sobre la base de datos. A intervalos menores de muestreo, se obtienen estadísticas más precisas, pero que requieren un costo mayor en tiempo de CPU y memoria. Es por eso que la mayoría de las implementaciones comerciales del Recolector de Estadísticas permite una configuración mediante parámetros externos al DBMS.

Básicamente, se recolectan en forma dinámica Estadísticas de varios objetos en una base de datos: Tablas, Columnas, Índices, etc. Esta información está disponible tanto para el Optimizador (interno) como para el usuario externo que quiera consultarla.

Arquitectura:

La finalidad del Recolector de Estadísticas es proveer información al Optimizador acerca de los datos a procesar. Hay dos métricas principales que se tienen en cuenta con alta prioridad: la **Cardinalidad** de una o más relaciones, y la **Selectividad** en una tabla.

Llamamos **"Cardinalidad"** al número de tuplas (filas) devueltas (o que se espera se devuelvan) por una operación en un Plan de Ejecución determinado. El Optimizador considera como **"Baja Cardinalidad"** al número de valores distintos en una columna, en relación al número total de filas.

El concepto de **"Selectividad"**, en cambio, nos indica la proporción del resultado de una consulta sobre la cantidad de filas de una relación. La **Selectividad** se mide con un valor entre 0 y 1. Selectividad = "0" indica que no se encontró ninguna fila que responda a la query, mientras que si obtenemos Selectividad = "1", el motor estaría devolviendo todas las filas de dicha relación. En la mayor parte de los casos, la Selectividad será un valor entre 0 y 1, donde un valor más cercano al "0" indica una baja proporción de filas devueltas, y lo opuesto a valores cercanos al "1".

Para ver en contexto estos dos indicadores y cómo estos influyen en la elección del Plan de Ejecución que hace el Optimizador, veamos un ejemplo sobre un DBMS real – en este caso, Microsoft SQL Server 2012:

- *Para comenzar, crearemos una nueva tabla temporal en la BD TempDB:*

```
USE TempDB
GO

IF (SELECT OBJECT_ID('Test1')) IS NOT NULL
      DROP TABLE dbo.Test1;
GO

CREATE TABLE dbo.Test1 (C1 INT, C2 INT IDENTITY);
```

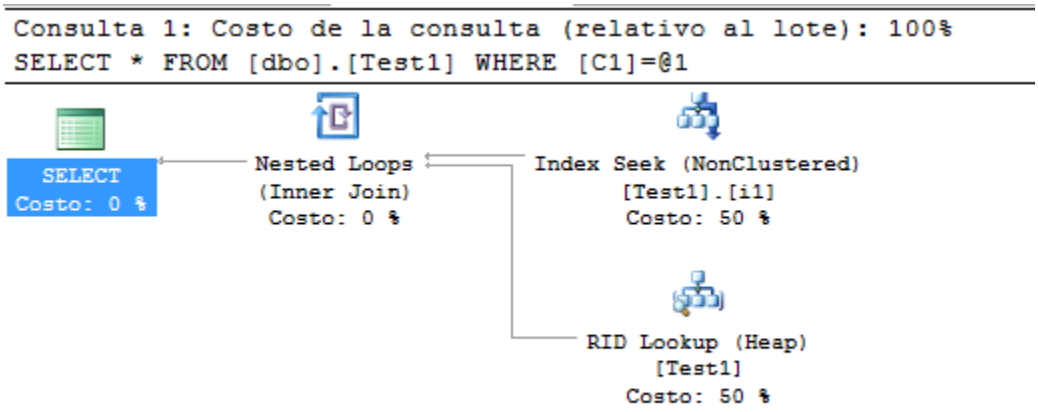
- *A continuación, insertaremos 1500 filas con números al azar, y crearemos un nuevo Indice No Agrupado:*

```
SELECT TOP 1500
      IDENTITY( INT,1,1 ) AS n
      INTO #Nums
      FROM Master.dbo.SysColumns sC1,
      Master.dbo.SysColumns sC2;

INSERT INTO dbo.Test1
      (C1)
      SELECT n
      FROM #Nums;

DROP TABLE #Nums;

CREATE NONCLUSTERED INDEX i1 ON dbo.Test1 (C1);
GO
```

- *Finalmente, ejecutaremos una consulta:*

```
SELECT * FROM dbo.Test1
      WHERE C1 = 2;
GO
```

- *El Plan de Ejecución obtenido será como el siguiente:*



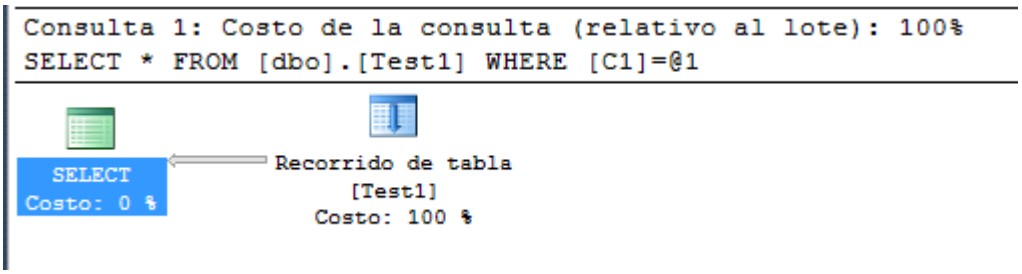
- *Obsérvese que la consulta sólo involucró a un ínfimo porcentaje de todas las filas de la relación (1 sobre 1500), por lo que el Optimizador, correctamente, elige como estrategia el uso del Índice No Agrupado.*
- *Ejecutemos ahora el siguiente script, que agrega 1500 nuevas filas a nuestra tabla:*

```
SELECT TOP 1500
       IDENTITY( INT,1,1 ) AS n
       INTO #Nums
       FROM Master.dbo.SysColumns scl,
       Master.dbo.SysColumns sC2;

INSERT INTO dbo.Test1
       (C1)
       SELECT 2
       FROM #Nums;

DROP TABLE #Nums;
GO
```

- *Y volvamos a ejecutar la misma consulta:*

```
SELECT * FROM dbo.Test1
       WHERE C1 = 2;
GO
```

- *El Plan de Ejecución cambia como sigue:*



- *Aquí podemos ver que el Optimizador recurre a un recorrido de Tabla Secuencial, debido a una baja* ***Cardinalidad*** *y una alta* ***Selectividad*** *(más de la mitad de las filas cumplen con la condición del WHERE). Si consultamos los valores de las Estadísticas para este Índice, tenemos el siguiente* ***Histograma****:*

  DBCC SHOW_STATISTICS(Test1, i1);

  GO

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows
1	i1	Jul 18 2016 8:29PM	3000	3000	196	1	4	NO	NULL	3000

	All density	Average Length	Columns
1	0,0006666667	4	C1

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	1	0	1	0	1
2	2	0	1501	0	1
3	8	5	1	5	1
4	16	7	1	7	1
5	20	3	1	3	1
6	28	7	1	7	1

, donde podemos observar una ***Alta Selectividad*** *para el valor* ***C1 = 2 (****EQ_ROWS = 1501 >> EQ_ROWS = 1****), por lo que el Optimizador decide hacer un Escaneo de Tabla (Table Scan) en lugar de usar el Índice.*

Es así como el contar con Estadísticas adecuadas redunda en una mejor performance de la consulta en cuestión debido a una mejor decisión del Optimizador en qué **Camino de Acceso** elegir.

Patrones Relacionados:

El Patrón **"Recolector de Estadísticas"** puede ser implementado tomando como base el Patrón de Diseño **"Observer"** [GoF01]. En tal caso, existirá un **"Subject"** para cada elemento al que se le quiere realizar mediciones, como ser: Tablas, Columnas, Índices, etc., implementando el **"Recolector de Estadísticas"** un conjunto de **"Observers"**.

***

Patrón: Analizador de Costos

Problema:

La tarea principal del **Optimizador** de SQL es la elección de uno entre un conjunto posible de **Caminos de Acceso**, a partir de una consulta. Dicha elección, como vimos, se basa principalmente en información de ejecuciones anteriores (Estadísticas), así como en una **evaluación de los Costos** que involucra cada una de dichos **Planes de Ejecución** alternativos. Sin embargo, debido a la relativa complejidad de dicho proceso de elección, se hace necesario que el mismo sea ejecutado por un componente especializado, de tal forma que no se convierta en una carga sobre la performance del DBMS.

Fuerzas:

El **Analizador de Costos** recibe como entrada un **Árbol de Consulta (lógico),** que representa la petición SQL suministrada al DBMS, y produce como salida un **Camino de Acceso (físico)**. Dicha estructura está compuesta por un conjunto de **nodos** organizados **jerárquicamente**, por lo que intuitivamente podemos deducir la necesidad de un **Algoritmo recursivo** que vaya evaluando el costo estimado de ejecución de cada uno de dichos nodos. Este Algoritmo se ejecutaría sobre cada uno de los Planes de Ejecución alternativos, eligiéndose al final el **Camino de Acceso de menor costo (óptimo)**.

Sin embargo, en la práctica nos encontramos con algunos problemas:

- Se deben evaluar todas las combinaciones posibles. Esto parece obvio, pero en caso de optar por esta estrategia, tendremos tiempos de respuesta del orden exponencial para el Optimizador → Fuerzas: Tiempo de CPU y Escalabilidad del Optimizador.
- Es imperativo diferenciar entre **Planes de Ejecución Lógicos** (provenientes de reordenar el Árbol de Ejecución original, suministrado por el **Algebrizer**), de los **Caminos de Acceso (Planes de Ejecución Físicos),** que involucran una serie de decisiones con el fin de optimizar el acceso al motor de almacenamiento → Fuerzas: Costo de CPU y Memoria para realizar esta doble evaluación.
- Finalmente, se debe poder asignar a cada nodo individual evaluado un **Costo**, y realizar una **Agregación** para poder calcular el **Costo Total de cada subárbol** → Fuerzas: Tiempo de CPU necesario para ejecutar un Algoritmo Recursivo, agravado por la *gran probabilidad de que el Optimizador deba calcular más de una vez el Costo (individual o agregado) para una misma operación dentro de un Plan de Ejecución.*

Solución:

Delegar el mecanismo de Elección de Planes de Ejecución a un componente denominado **"Analizador de Costos"**. Este componente tendrá la responsabilidad de construir un conjunto de

Planes de Ejecución alternativos, para cada uno evaluar los distintos Caminos de Acceso que puedan responder a cada petición, y finalmente elegir el de menor Costo.
Para evitar un fenómeno de explosión combinatoria, así como evitar la doble evaluación que citamos anteriormente, los **Analizadores de Costos** implementan Algoritmos basados en la técnica de ***"Programación Dinámica"*** o alguna variante de la misma. Mediante dichos Algoritmos, este componente puede recorrer cada **Plan de Ejecución (lógico)**, evaluando los costos de cada posible orden en los JOINs, la posibilidad de reutilizar predicados SARGables, etc., generando así un reordenamiento del Árbol de Ejecución. La medición de estos valores se hace en base a la información de Estadísticas (Histograma, Cardinalidad y Selectividad), suministradas por el Patrón **"Recolector de Estadísticas"**. La evaluación del **Camino de Acceso (físico)** se hace en forma análoga, con la diferencia que aquí se tiene en cuenta los Costos Físicos de Acceso a cada recurso (CPU, Memoria, I/O).

Arquitectura:
El componente **"Analizador de Costos"** utiliza, en forma intensiva, Algoritmos basados en alguna variante de "Programación Dinámica", con la finalidad de elegir un Camino de Acceso óptimo. Esta variación se debe a que en Programación Dinámica clásica, a cada nodo se le calcula un Costo Óptimo, y se lo hace una única vez en todo el procedimiento. Para cada nodo pueden coexistir más de un Costo Óptimo, definido en una especie de "ranking de costos" que involucra no sólo el óptimo sino también uno o más de los llamados **"Órdenes Interesantes"** [Selinger01].
Un **"Orden Interesante"** es una propiedad que puede tener un nodo de producir un resultado que, a pesar de no ser el óptimo para el Camino de Acceso del cual es raíz (sub-árbol), pueda ser conveniente para operadores que estén más arriba en la jerarquía del Plan de Ejecución. Por ejemplo, el Analizador de Costos puede seleccionar el Camino de Acceso de menor costo, pero también considerar otro que, a pesar de tener asociado un costo mayor, devuelva una salida ordenada, la cual pueda ser aprovechada a continuación por un operador de **"Merge Join"**. Este tipo de heurísticas ayudan a converger en un ***Camino de Acceso óptimo para todo el Árbol de Ejecución***.

A continuación, analizaremos las consideraciones aplicables al Análisis de Costos para tres principales casos: consulta simple (1 sola tabla), JOIN, y Sub-consultas.3

- **Costos de Consultas Simples**
  El caso más simple a tener en cuenta es la optimización de una consulta que involucra una sola tabla. Al Analizador de Costos le interesa conocer:

  - ***Si se puede aplicar un Índice (Agrupado o No Agrupado):***

La posibilidad de utilizar un Índice como parte del Camino de Acceso implica el uso de operadores que usan la estructura básica de árbol **B-Tree** para acceder a un valor en particular, o en su defecto un recorrido en forma ordenada de los datos. En el primer caso, la operación de **"Index Seek"** es mucho más rápida que su contraparte **"Table Scan"**, mientras que un **"Index Scan"** no requiere un ordenamiento posterior de los datos, ya que se recorre el Índice en forma ordenada. En ambos casos, los Costos de estas operaciones son de **"Orden Logarítmico"** cuando se utilizan Índices, mientras que en otro caso los Costos pueden ser lineales o exponenciales.

- ***Si hay que hacer un Lookup:***

  En los **Índices No Agrupados** (Non Clustered), el árbol de indexación solo contiene información de las claves por las que dicho Índice fue definido (esto contrasta con los **Índices Agrupados** (Clustered), donde la estructura de árbol físicamente ordena las Páginas de Datos). Esto ocasiona que los Caminos de Acceso que utilicen Índices No Agrupados requieran una operación adicional para obtener datos en el resto de las columnas no indexadas: la operación de **"Lookup"**.

- ***Si el Predicado en el WHERE es SARGable:***

  Denominamos como **"SARGable"** a todo Predicado que permita la utilización efectiva de uno o más Índices que se encuentren definidos en una tabla determinada.

- ***Si se debe ordenar el resultado:***

  Esta consideración nos indica si es necesario que el resultado tenga un ***"Orden Interesante"*** [Selinger01], o en su defecto si se debe materializar la salida de un operador para ser tomada como respuesta a la petición SQL original.

- **Costos de la operación de JOIN**

  Antes de definir los operadores de JOIN que pueden ser parte de un Camino de Acceso, el Optimizador puede realizar un cambio en el orden de dichas operaciones ante la detección de JOINs múltiples (más de dos tablas involucradas). Este tipo de optimización suele realizarse como primera medida al iniciar la ejecución del Análisis de Costos, y se utiliza en forma exhaustiva el conjunto de Estadísticas del que dispone el DBMS. Este tipo de análisis tiene la característica de no ser exhaustivo, sino que sólo se evalúa un subconjunto de los posibles órdenes en los JOINs, eligiéndose el de menor costo. El

análisis que sigue se basa en la asociatividad de las operaciones de JOIN, lo que significa que si podemos optimizar cada JOIN entre dos tablas, entonces estaremos en condiciones de extender nuestros resultados a "n" relaciones.
Los operadores físicos que implementan la operación de JOIN entre dos tablas son: Nested Loop Join, Merge Join, y Hash Join. A continuación detallaremos las características de cada uno de ellos.

Nested Loop Join:

Es la manera más común de unir dos relaciones. Básicamente, define una tabla como **"Outer"** y la otra como **"Inner"**. A continuación, un bucle anidado recorre cada registro de la tabla "Outer", ejecutando una consulta también secuencial sobre la tabla "Inner". La condición de JOIN actúa en este caso como filtro, de tal manera que no se produzca una explosión combinatoria (todos contra todos).
El Costo de este operador depende en gran medida de la cardinalidad de cada tabla. Podemos obtener una gran variabilidad de performance con sólo invertir el orden de las tablas "Outer" e "Inner", sobre todo en casos donde una tabla contenga relativamente más filas que la otra.

Merge Join:

El Optimizador suele elegir este tipo de JOIN cuando cada una de las tablas participantes defina un **"orden interesante"**. Esto puede requerir la definición de Índices que provean dicho orden, o en su defecto la disponibilidad de espacio en los buffers para un ordenamiento temporal de los datos. El estar sincronizados por un orden definido, implica que no se requieran escaneos obligatorios de todo el conjunto de filas para cada tabla.
La eficiencia de este Algoritmo depende en gran medida de la disponibilidad de dichos **"órdenes interesantes"**.

Hash Join:

Este tercer operador se utiliza cuando se detecta una gran diferencia en la cardinalidad de las tablas participantes. Básicamente, se elige como tabla "Inner" a la relación con menor cantidad de filas, y se le calcula un **"Hash"** para cada valor a Joinear. La tabla restante, "Outer", con muchas más filas, se recorre una sola vez, asociando los valores según el Predicado de JOIN.
Dadas estas condiciones, este operador ofrece una buena performance en relación a los dos anteriores. Sin embargo, la complejidad de la operación de "Hash", hace que sea elegido por el Optimizador sólo en contadas ocasiones.

El Costo de este operador radica en gran medida en la carga de CPU requerida para calcular los valores "hash" de cada tupla en la relación "Inner".
Finalmente, debemos notar que, en algunos DBMS puntuales, la opción de **"Hash Join"** no esté implementada.

En general, el Costo de la operación de JOIN suele ser crítico para cada uno de los recursos a tener en cuenta (CPU, Memoria e I/O). La alternativa más simple y directa es sin duda la operación de **"Nested Loop"**, por lo que observamos que todos los optimizadores de DBMS comerciales lo prefieren a la hora de elegir un Camino de Acceso. Sin embargo, factores adicionales como la cardinalidad y selectividad, disponibilidad de un conjunto de Estadísticas actualizado, o simplemente la existencia de modificadores del SQL original que fuercen la elección de uno u otro método de acceso (Hints), pueden inclinar la balanza a favor de alguno de las dos restantes estrategias.

- **Costos de Subconsultas**

  Una subconsulta se compone básicamente de dos partes: una consulta principal, denominada "Externa" (Outer), y otra dependiente, denominada "Interna" (Inner). Podemos entonces deducir que el Costo de Ejecución de una subconsulta, será la suma de los Costos de Ejecucion de la consulta "Inner" más la de la "Outer". Esto será así siempre y cuanto la consulta Interna no utilice valores dependientes de la Externa, es decir, mientras sea una **"Consulta NO Correlacionada"**.

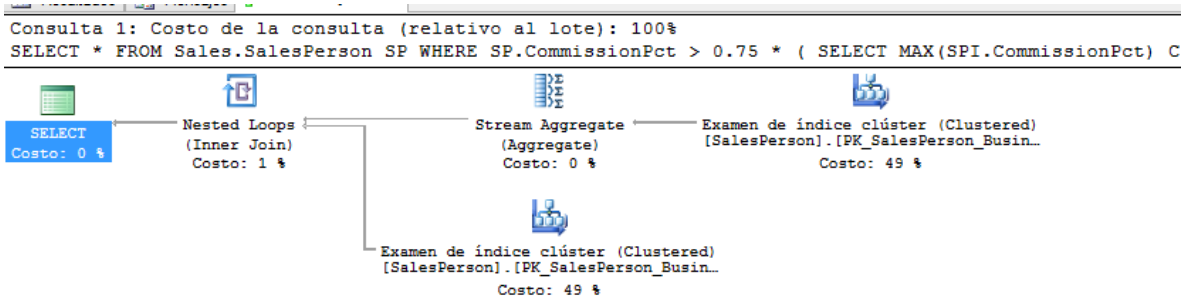
  Decimos que una Subconsulta es **"Correlacionada"**, cuando la consulta Interna (Inner) utiliza valores provenientes de la Externa (Outer), devolviendo un resultado que termina utilizando la Consulta Externa (Outer). En este caso, la consulta Interna (Inner) debe ser re-evaluada para cada fila de la Externa, por lo que los Costos de Ejecución dejan de ser lineales.

  Analicemos la siguiente consulta no correlacionada, que devuelve las Ventas realizadas cuya Comisión supere el 75% de la mayor comisión registrada:

```
USE AdventureWorks2012
GO

SELECT *
      FROM Sales.SalesPerson SP
      WHERE SP.CommissionPct > 0.75 * (
            SELECT MAX(SPI.CommissionPct) CommissionPct
```

```
                    FROM Sales.SalesPerson SPI
                    WHERE SPI.CommissionPct IS NOT NULL);
GO
```

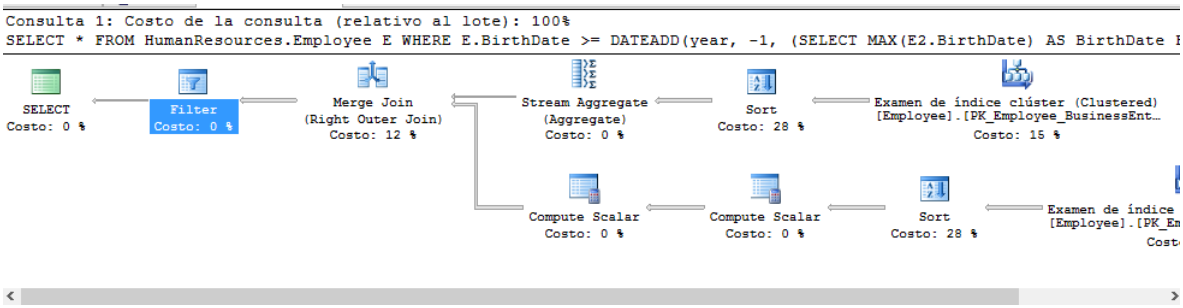
Si observamos el Plan de Ejecución que genera SQL Server 2012, veremos algo como sigue:



El Plan de Ejecución indica como Camino de Acceso un operador de **"Nested Loops"**, que significa que el Optimizador transformó la subconsulta en un JOIN equivalente.
Veamos ahora otro ejemplo, en este caso una Subconsulta Correlacionada: retornar los datos de los Empleados cuya Fecha de Nacimiento no sea anterior a un Año en relación a la Fecha de Nacimiento del más antiguo de los Empleados en su puesto.

```
USE AdventureWorks2012
GO

SELECT *
      FROM HumanResources.Employee E
      WHERE E.BirthDate >= DATEADD(year, -1,
            (SELECT MAX(E2.BirthDate) AS BirthDate
                  FROM HumanResources.Employee E2
                  WHERE E2.JobTitle = E.JobTitle));
GO
```

El Plan de Ejecución sigue basándose en un JOIN de tipo **"Merge Join"**, pero también se agregan nuevos operadores de Filtro y Ordenamiento:



Para finalizar, debemos decir que las mismas consideraciones para la estimación de Costos tanto en consultas Correlacionadas como en no Correlacionadas, pueden extenderse para un anidamiento de más niveles (es decir, una subconsulta de otra subconsulta interna).

Patrones Relacionados:

Como vimos anteriormente, los Analizadores de Costos utilizan alguna variante de los Algoritmos de "Programación Dinámica". Esta técnica se basa principalmente en la implementación del Patrón **"Memoization"** para reutilizar resultados intermedios en la estimación de costos para los Caminos de Acceso elegidos: tanto **[Norvig01]**, en relación acerca de la implementación de Parsers en LISP, y **[Michie01]** en su trabajo acerca del uso de "Memoizadores" en Aprendizaje Automático, tratan sobre dicha técnica**.** Microsoft SQL Server 2008 y sus versiones posteriores, utilizan oficialmente dicho Patrón en la implementación del Optimizador de SQL, denominándolo **"Memo" [Delaney01]**. Por otra parte, algunos DBMSs implementan otras variantes del Analizador de Costos mediante **"Algoritmos Adaptativos"** o incluso **"Algoritmos Genéticos"**, como es el caso del Optimizador de PostgreSQL **[ZFong01]**.

***

Bibliografía

[Athanass01] – Athanassoulis, Manos, "Designing Access Methods: The RUM Conjecture".

[Buschmann01] – Buschmann, Frank, "Pattern Oriented Software Architectures".

[Cormen01] – Cormen, Thomas, "Introduction to Algorithms".

[Delaney01] – Delaney, Karen, "Microsoft SQL Server 2008 Internals".

[GoF01] – Gamma, Helm, Johnson, Vlissides, "Patrones de Diseño".

[Jarke01] – Jarke, Matthias, Koch, Jürgen, "Query Optimization in Database Systems". Computing Surveys, Vol. 16, No. 2, June 1984.

[Mannino01] – Mannino, Michael, Chu, Paicheng, Sager, Thomas, "Statistical Profile Estimation in Database Systems".

[Michie01] - Michie, Donald, "Memo Functions and Machine Learning".

[Norvig01] - Norvig, Peter. 1991, "Techniques for automatic memoization with applications to context-free parsing".

[Selinger01] – Selinger, Astrahan, Chamberlain, “Access Path Selection in a Relational Database Management System”.
[ZFong01] – “The design and implementation of the POSTGRES query optimizer”.

***